








Answer Set Programming Made Easy

Jorge Fandinno^{1,2} , Seemran Mishra² , Javier Romero² , and Torsten Schaub²  

¹ University of Nebraska at Omaha, Omaha, USA

² University of Potsdam, Potsdam, Germany

smishra@uni-potsdam.de, torsten@cs.uni-potsdam.de

Abstract. We take up an idea from the folklore of Answer Set Programming (ASP), namely that choices, integrity constraints along with a restricted rule format is sufficient for ASP. We elaborate upon the foundations of this idea in the context of the logic of Here-and-There and show how it can be derived from the logical principle of extension by definition. We then provide an austere form of logic programs that may serve as a normalform for logic programs similar to conjunctive normalform in classical logic. Finally, we take the key ideas and propose a modeling methodology for ASP beginners and illustrate how it can be used.

1 Introduction

Many people like Answer Set Programming (ASP [20]) because its declarative approach frees them from expressing any procedural information. In ASP, neither the order of rules nor the order of conditions in rule antecedents or consequents matter and thus leave the meaning of the overall program unaffected. Although this freedom is usually highly appreciated by ASP experts, sometimes laypersons seem to get lost without any structural guidance when modeling in ASP.

We address this issue in this (preliminary) paper and develop a methodology for ASP modeling that targets laypersons, such as biologists, economists, engineers, and alike. As a starting point, we explore an idea put forward by Ilkka Niemelä in [25], although already present in [10, 16] as well as the neighboring area of Abductive Logic Programming [7, 9]. To illustrate it, consider the logic program encoding a Hamiltonian circuit problem in Listing 1.1. Following good practice in ASP, the problem is separated into the specification of the problem instance in lines 1–3 and the problem class in lines 5–10. This strict separation, together with the use of facts for problem instances, allows us to produce uniform¹ and elaboration tolerant² specifications. Building upon the facts of the problem instance, the actual encoding follows the guess-define-check methodology of ASP. A solution candidate is guessed in Line 5, analyzed by auxiliary definitions in Line 6 and 7, and finally checked through integrity constraints in lines 8–10.

A closer look reveals even more structure in this example. From a global perspective, we observe that the program is partitioned into facts, choices, rules, and integrity

¹ A problem encoding is *uniform*, if it can be used to solve all its problem instances.

² A formalism is *elaboration tolerant* if it is convenient to modify a set of facts expressed in the formalism to take into account new phenomena or changed circumstances [24].

```

1  node(1..4).          start(1).
2  edge(1,2). edge(2,3). edge(2,4). edge(3,1).
3  edge(3,4). edge(4,1). edge(4,3).
4
5  { hc(V,U) } :- edge(V,U).
6  reached(V) :- hc(S,V), start(S).
7  reached(V) :- reached(U), hc(U,V).
8  :- node(V), not reached(V).
9  :- hc(V,U), hc(V,W), U!=W.
10 :- hc(U,V), hc(W,V), U!=W.

```

Listing 1.1. A logic program for a Hamiltonian circuit problem

constraints, and in this order. From a local perspective, we note moreover that the predicates in all rule antecedents are defined beforehand. This structure is not arbitrary and simply follows the common practice that concept formation is done linearly by building concepts on top of each other. Moreover, it conveys an intuition on how a solution is formed. Importantly, such an arrangement of rules is purely methodological and has no impact on the meaning (nor the performance³) of the overall program. From a logical perspective, it is interesting to observe that the encoding refrains from using negation explicitly, except for the integrity constraints. Rather this is hidden in Line 5, where the choice on $hc(V, U)$ amounts to the disjunction $hc(V, U) \vee \neg hc(V, U)$, an instance of the law of the excluded middle. Alternatively, $hc(V, U)$ can also be regarded as an abducible that may or may not be added to a program, as common in Abductive Logic Programming.

Presumably motivated by similar observations, Ilkka Niemelä already argued in [25] in favor of an ASP base language based on choices, integrity constraints, and stratified negation.⁴ We also have been using such an approach when initiating students to ASP as well as teaching laypersons. Our experience has so far been quite positive and we believe that a simple and more structured approach helps to get acquainted with posing constraints in a declarative setting.

We elaborate upon this idea in complementary ways. First of all, we lift it to a logical level to investigate its foundations and identify its scope. Second, we want to draw on this to determine a syntactically restricted subclass of logic programs that still warrants the full expressiveness of traditional ASP. Such a subclass can be regarded as a normalform for logic programs in ASP. This is also interesting from a research perspective since it allows scientists to initially develop their theories in a restricted setting without regarding all corner-cases emerging in a full-featured setting. And last but not least, inspired by this, we want to put forward a simple and more structured modeling methodology for ASP that aims at beginners and laypersons.

³ Shuffling rules in logic programs has an effect on performance since it affects tie-breaking during search; this is however unrelated to the ordering at hand.

⁴ This concept eliminates the (problematic) case of recursion through negation.

2 Background

The logical foundations of ASP rest upon the logic of Here-and-There (HT [17]) along with its non-monotonic extension, Equilibrium Logic [26].

We start by defining the monotonic logic of *Here-and-There* (HT). Let \mathcal{A} be a set of atoms. A *formula* φ over \mathcal{A} is an expression built with the grammar:

$$\varphi ::= a \mid \perp \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \rightarrow \psi$$

for any atom $a \in \mathcal{A}$. We also use the abbreviations: $\neg\varphi \stackrel{\text{def}}{=} (\varphi \rightarrow \perp)$, $\top \stackrel{\text{def}}{=} \neg\perp$, and $\varphi \leftrightarrow \psi \stackrel{\text{def}}{=} (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$. Given formulas φ , α and β , we write $\varphi[\alpha/\beta]$ to denote the uniform substitution of all occurrences of formula α in φ by β . This generalizes to the replacement of multiple formulas in the obvious way. As usual, a *theory* over \mathcal{A} is a set of formulas over \mathcal{A} . We sometimes understand finite theories as the conjunction of their formulas.

An *interpretation* over \mathcal{A} is a pair $\langle H, T \rangle$ of atoms (standing for “here” and “there”, respectively) satisfying $H \subseteq T \subseteq \mathcal{A}$. An interpretation is *total* whenever $H = T$. An interpretation $\langle H, T \rangle$ *satisfies* a formula φ , written $\langle H, T \rangle \models \varphi$, if the following conditions hold:

$$\begin{aligned} \langle H, T \rangle \models p & \quad \text{if } p \in H \\ \langle H, T \rangle \models \varphi \wedge \psi & \quad \text{if } \langle H, T \rangle \models \varphi \text{ and } \langle H, T \rangle \models \psi \\ \langle H, T \rangle \models \varphi \vee \psi & \quad \text{if } \langle H, T \rangle \models \varphi \text{ or } \langle H, T \rangle \models \psi \\ \langle H, T \rangle \models \varphi \rightarrow \psi & \quad \text{if } \langle H', T \rangle \not\models \varphi \text{ or } \langle H', T \rangle \models \psi \text{ for both } H' \in \{H, T\} \end{aligned}$$

A formula φ is *valid*, written $\models \varphi$, if it is satisfied by all interpretations. An interpretation $\langle H, T \rangle$ is a *model* of a theory Γ , written $\langle H, T \rangle \models \Gamma$, if $\langle H, T \rangle \models \varphi$ for all $\varphi \in \Gamma$.

Classical entailment is obtained via the restriction to total models. Hence, we define the classical satisfaction of a formula φ by an interpretation T , written $T \models \varphi$, as $\langle T, T \rangle \models \varphi$.

A total interpretation $\langle T, T \rangle$ is an *equilibrium model* of a theory Γ if $\langle T, T \rangle$ is a model of Γ and there is no other model $\langle H, T \rangle$ of Γ with $H \subset T$. In that case, we also say that T is a *stable model* of Γ . We denote the set of all stable models of Γ by $SM[\Gamma]$ and use $SM_V[\Gamma] \stackrel{\text{def}}{=} \{T \cap V \mid T \in SM[\Gamma]\}$ for their projection onto some vocabulary $V \subseteq \mathcal{A}$.

Since ASP is a non-monotonic formalism, it may happen that two different formulas share the same equilibrium models but behave differently in different contexts. The concept of *strong equivalence* captures the idea that two such formulas have the same models regardless of any context. More precisely, given two theories Γ and Π and a set $V \subseteq \mathcal{A}$ of atoms, we say that Γ and Π are *V-strongly equivalent* [2], written $\Gamma \cong_V \Pi$, if $SM_V[\Gamma \cup \Delta] = SM_V[\Pi \cup \Delta]$ for any theory Δ over \mathcal{A}' such that $\mathcal{A}' \subseteq V$. For formulas φ and ψ , we write $\varphi \cong_V \psi$ if $\{\varphi\} \cong_V \{\psi\}$.

A *rule* is a (reversed) implication of the form

$$l_1 \vee \dots \vee l_m \leftarrow l_{m+1} \wedge \dots \wedge l_n \tag{1}$$

where each l_i is a literal, that is, either an atom or a negated atom, for $1 \leq i \leq n$. If $n = 1$, we refer to the rule as a *fact* and write it as l_1 by dropping the trailing implication symbol. A rule is said to be *normal* whenever $m = 1$ and l_1 is an atom. A negation-free normal rule is called *definite*. An *integrity constraint* is a rule with $m = 0$ and equivalent to $\perp \leftarrow l_{m+1} \wedge \cdots \wedge l_n$. Finally, the law of the excluded middle $a \vee \neg a$ is often represented as $\{a\}$ and called a *choice*. Accordingly, a rule with a choice on the left-hand side is called a *choice rule*. A *logic program* is a set of rules. It is called *normal*, if it consists only of normal rules and integrity constraints, and *definite* if all its rules are definite.

3 Logical Foundations

We begin by investigating the logical underpinnings of the simple format of logic programs discussed in the introductory section. Although the discussion of the exemplary logic program has revealed several characteristic properties, not all of them can be captured in a logical setting, such as order related features. What remains is the division of the encoding into facts, rules, choices, and integrity constraints. In logical terms, the first two amount to negation-free formulas, choices are instances of the law of the excluded middle, and finally integrity constraints correspond to double-negated formulas in HT. While the first two types of formulas are arguably simpler because of their restricted syntax, the latter's simplicity has a semantic nature and is due to the fact that in HT double negated formulas can be treated as in classical logic.

In what follows, we show that any formula can be divided into a conjunction of corresponding subformulas. This conjunction is strongly equivalent (modulo the original vocabulary) to the original formula and the translation can thus also be applied to substitute subformulas. Interestingly, the resulting conjunction amounts to a conservative extension of the original formula and the underlying translation can be traced back to the logical principle of extension by definition, as we show below.

To this end, we associate with each formula φ over \mathcal{A} a new propositional atom x_φ . We then consider defining axioms of the form $(x_\varphi \leftrightarrow \varphi)$. We can now show that replacing any subformula φ by x_φ while adding a corresponding defining axiom amounts to a conservative extension of ψ .⁵

Proposition 1. *Let ψ and φ be formulas over \mathcal{A} and $x_\varphi \notin \mathcal{A}$.*

Then, $\psi \cong_{\mathcal{A}} (\psi[\varphi/x_\varphi] \wedge (\varphi \leftrightarrow x_\varphi))$.

Moreover, we get a one-to-one correspondence between the stable models of both formulas.

Proposition 2. *Let ψ and φ be formulas over \mathcal{A} and $x_\varphi \notin \mathcal{A}$.*

1. *If $T \subseteq \mathcal{A}$ is a stable model of ψ , then $T \cup \{x_\varphi \mid T \models \varphi\}$ is a stable model of $(\psi[\varphi/x_\varphi] \wedge (\varphi \leftrightarrow x_\varphi))$.*
2. *If $T \subseteq (\mathcal{A} \cup \{x_\varphi\})$ is a stable model of $(\psi[\varphi/x_\varphi] \wedge (\varphi \leftrightarrow x_\varphi))$, then $T \cap \mathcal{A}$ is a stable model of ψ .*

⁵ An extended version of the paper including all proofs can be found here: <https://arxiv.org/abs/2111.06366>.

Clearly, the above results generalize from replacing and defining a single subformula φ to several such subformulas.

With this, we can now turn our attention to negated subformulas: Given a formula ψ , let $N(\psi)$ stand for the set of all maximal negated subformulas occurring in ψ . This leads us to the following variant of Proposition 1.

Corollary 1. *Let ψ be a formula over \mathcal{A} and $x_\varphi \notin \mathcal{A}$.*

Then, $\psi \cong_{\mathcal{A}} \psi[\varphi/x_\varphi \mid \varphi \in N(\psi)] \wedge \bigwedge_{\varphi \in N(\psi)} (\varphi \leftrightarrow x_\varphi)$.

Given that we exclusively substitute negated subformulas, we can actually treat the defining axiom as in classical logic. This is because in HT, we have $\langle H, T \rangle \models \neg\varphi$ iff (classically) $T \models \neg\varphi$. The classical treatment of the defining axiom is then accomplished by replacing $(\varphi \leftrightarrow x_\varphi)$ by $\neg\neg(\varphi \leftrightarrow x_\varphi)$ and $(\neg x_\varphi \vee x_\varphi)$. This results in the following decomposition recipe for formulas.

Definition 1. *Let ψ be a formula over \mathcal{A} and $x_\varphi \notin \mathcal{A}$.*

Then, we define

$$\psi^* = \psi[\varphi/x_\varphi \mid \varphi \in N(\psi)] \wedge \bigwedge_{\varphi \in N(\psi)} (\neg x_\varphi \vee x_\varphi) \wedge \bigwedge_{\varphi \in N(\psi)} \neg\neg(\varphi \leftrightarrow x_\varphi).$$

Example 1. Let ψ be $\neg a \rightarrow b \vee \neg\neg(c \wedge \neg d)$. Then,

$$\begin{aligned} N(\psi) &= \{\neg a, \neg\neg(c \wedge \neg d)\} \\ \psi^* &= (x_{\neg a} \rightarrow b \vee x_{\neg\neg(c \wedge \neg d)}) \wedge \\ &\quad (x_{\neg a} \vee \neg x_{\neg a}) \wedge (x_{\neg\neg(c \wedge \neg d)} \vee \neg x_{\neg\neg(c \wedge \neg d)}) \\ &\quad \neg\neg(\neg a \leftrightarrow x_{\neg a}) \wedge \neg\neg(\neg\neg(c \wedge \neg d) \leftrightarrow x_{\neg\neg(c \wedge \neg d)}) \end{aligned}$$

With the translation from Definition 1, we obtain an analogous conservative extension result as above.

Theorem 1. *Let ψ be a formula over \mathcal{A} .*

Then, we have $\psi \cong_{\mathcal{A}} \psi^$.*

In analogy to Proposition 2, we get a one-to-one correspondence between the stable models of both formulas.

Theorem 2. *Let ψ be a formula over \mathcal{A} .*

1. *If $T \subseteq \mathcal{A}$ is a stable model of ψ , then $T \cup \{x_\varphi \mid \varphi \in N(\psi) \text{ and } T \models \varphi\}$ is a stable model of ψ^* .*
2. *If $T \subseteq (\mathcal{A} \cup \{x_\varphi \mid \varphi \in N(\psi)\})$ is a stable model of ψ^* , then $T \cap \mathcal{A}$ is a stable model of ψ .*

For instance, $\{b\}$ is a stable model of the formula $\psi = \neg a \rightarrow b \vee \neg\neg(c \wedge \neg d)$ from Example 1. From Theorem 1, $\{x_{\neg a}, b\}$ is a stable model of ψ^* . Conversely, from the stable model $\{x_{\neg a}, b\}$ of ψ^* , we get the stable model $\{b\}$ of ψ by dropping the new atoms.

4 Austere Answer Set Programming

In this section, we restrict the application of our formula translation to logic programs. Although we focus on normal programs, a similar development with other classes of logic programs, like disjunctive ones, can be done accordingly.

For simplicity, we write \bar{a} instead of $x_{\neg a}$ for $a \in \mathcal{A}$ and let $\{\bar{a}\}$ stand for $\bar{a} \vee \neg\bar{a}$. Note that, for a rule r as in (1), the set $N(r)$ consists of negative literals only. The next two definitions specialize our translation of formulas to logic programs.

Definition 2. Let r be a rule over \mathcal{A} as in (1) with $m \geq 1$.

Then, we define

$$r^* = r[\neg a/\bar{a} \mid \neg a \in N(r)] \cup \bigcup_{\neg a \in N(r)} \{\{\bar{a}\} \leftarrow\} \cup \bigcup_{\neg a \in N(r)} \left\{ \begin{array}{l} \leftarrow a \wedge \bar{a} \\ \leftarrow \neg a \wedge \neg\bar{a} \end{array} \right\}$$

Definition 3. Let P be a logic program over \mathcal{A} . Then, $P^* = \bigcup_{r \in P} r^*$.

This translation substitutes negated literals in rule bodies with fresh atoms and adds a choice rule along with a pair of integrity constraints providing an equivalence between the eliminated negated body literals and the substituted atoms.

By applying the above results in the setting of logic programs, we get that a logic program and its translation have the same stable models when restricted to the original vocabulary.

Corollary 2. Let P be a logic program over \mathcal{A} .

Then, we have $P \cong_{\mathcal{A}} P^*$

In other words, every stable model of a logic program can be extended to a stable model of its translation and vice versa.

Corollary 3. Let P be a logic program over \mathcal{A} .

1. If $T \subseteq \mathcal{A}$ is a stable model of P , then $T \cup \{\bar{a} \mid \neg a \in N(P) \text{ and } a \notin T\}$ is a stable model of P^* .
2. $T \subseteq (\mathcal{A} \cup \{\bar{a} \mid \neg a \in N(P)\})$ is a stable model of P^* , then $T \cap \mathcal{A}$ is a stable model of P .

For illustration, consider the following example.

Example 2. Consider the normal logic program P :

$$\begin{array}{l} a \leftarrow \\ b \leftarrow \neg c \\ c \leftarrow \neg b \\ d \leftarrow a \wedge \neg c \end{array}$$

Then, P^* is:

$$\begin{array}{lll} a \leftarrow & \{\bar{b}\} \leftarrow & \{\bar{c}\} \leftarrow \\ b \leftarrow \bar{c} & \leftarrow b \wedge \bar{b} & \leftarrow c \wedge \bar{c} \\ c \leftarrow \bar{b} & \leftarrow \neg b \wedge \neg\bar{b} & \leftarrow \neg c \wedge \neg\bar{c} \\ d \leftarrow a \wedge \bar{c} & & \end{array}$$

The stable models of P are $\{a, b, d\}$ and $\{a, c\}$ and the ones of P^* are $\{a, b, d, \bar{c}\}$ and $\{a, c, \bar{b}\}$, respectively.

The example underlines that our translation maps normal rules to definite ones along with choices and pairs of integrity constraints. In other words, it can be seen as a means for expressing normal logic programs in the form of programs with facts, definite rules, choice rules and integrity constraints over an extended vocabulary. We call this class of programs *austere logic programs*, and further elaborate upon them in the following.

4.1 Austere Logic Programs

We define austere logic programs according to the decomposition put forward in the introduction.

Definition 4 (Austere logic program). *An austere logic program is a quadruple (F, C, D, I) consisting of a set F of facts, a set C of choices,⁶ a set D of definite rules, and a set I of integrity constraints.*

A set of atoms is a stable model of an austere logic program, if it is a stable model of the union of all four components.

In view of the above results, austere logic programs can be regarded as a normal form for normal logic programs.

Corollary 4. *Every normal logic program can be expressed as an austere logic program and vice versa.*

The converse follows from the fact that choice rules are expressible by a pair of normal rules [27].

In fact, the (instantiation of) Listing 1.1 constitutes an austere logic program. To see this observe that

- lines 1–3 provide facts, F , capturing the problem instance, here giving the specification of a graph;
- Line 5 provides choices, C , whose instantiation is derived from facts in the previous lines. Grounding expands this rule to several plain choice rules with empty bodies;
- lines 5–6 list definite rules, D , defining (auxiliary) predicates used in the integrity constraints;
- finally, integrity constraints, I , are given in lines 7–9, stating conditions that solutions must satisfy.

This example nicely illustrates a distinguishing feature of austere logic programs, namely, the *compartmentalization* of the program parts underlying ASP’s guess-define-check encoding methodology (along with its strict separation of instance and encoding): The problem instance is described by means of

- the facts in F

and the problem encoding confines

- non-deterministic choices to C ,
- the deterministic extension of the taken decisions to D , and

⁶ That is, choice rules without body literals.

- the test of the obtained extension to I .

This separation also confines the sources of multiple or non-existing stable models to well-defined locations, namely, C and I , respectively (rather than spreading them over several circular rules; see below). As well, the rather restricted syntax of each compartment gives rise to a very simple operational semantics of austere logic programs, as we see in the next section.

4.2 Operational Semantics

In our experience, a major factor behind the popularity of the approach sketched in the introductory section lies in the possibility to intuitively form stable models along the order of the rules in a program. In fact, the simple nature of austere logic programs provides a straightforward scheme for computing stable models by means of the well-known immediate consequence operator, whose iteration mimics this proceeding. Moreover, the simplicity of the computation provides the first evidence of the value of austere logic programs as a normalform.

The operational semantics of austere logic programs follows ASP's *guess-define-check* methodology. In fact, the only non-determinism in austere logic programs is comprised of choice rules. Hence, once choices are made, we may adapt well-known deterministic bottom-up computation techniques for computing stable models. However, the results of this construction provide merely candidate solutions that still need to satisfy all integrity constraints. If this succeeds, they constitute stable models of the austere program.

Let us make this precise for an austere logic program (F, C, D, I) in what follows. To make choices and inject them into the bottom-up computation, we translate the entire set of choices, C , into a set of facts:

$$F_C = \{a \leftarrow \mid \{a\} \leftarrow \in C\}$$

A subset of F_C , the original facts F , along with the definite program D are then passed to a corresponding consequence operator that determines a unique stable model candidate. More precisely, the T_P operator of a definite program P is defined for an interpretation X as follows [23]:

$$T_P(X) = \{l_1 \mid (l_1 \leftarrow l_{m+1} \wedge \dots \wedge l_n) \in P, X \models l_{m+1} \wedge \dots \wedge l_n\}$$

With this, the candidate solutions of an austere program can be defined.

Definition 5. *Let (F, C, D, I) be an austere logic program over \mathcal{A} .*

We define a set $X \subseteq \mathcal{A}$ of atoms as a candidate stable model of (F, C, D, I) , if X is the least fixpoint of $T_{F \cup C' \cup D}$ for some $C' \subseteq F_C$.

The existence of the least fixpoint is warranted by the monotonicity of $T_{F \cup C' \cup D}$ [23]. Similar to traditional ASP, several candidate models are obtained via the different choices of C' .

While the choice of C' constitutes the *guess* part and the definite rules in D the *define* part of the approach, the *check* part is accomplished by the integrity constraints in I .

Proposition 3. *Let (F, C, D, I) be an austere logic program over \mathcal{A} and $X \subseteq \mathcal{A}$.*

Then, X is a stable model of (F, C, D, I) iff X is a candidate stable model of (F, C, D, I) such that $X \models I$.

We illustrate the computation of stable models of austere logic programs in the following example.

Example 3. Consider the austere logic program P

$$\begin{aligned} a &\leftarrow \\ \{b\} &\leftarrow \\ c &\leftarrow b \\ &\leftarrow a \wedge \neg c \end{aligned}$$

We get the candidate stable models $\{a, b, c\}$ and $\{a\}$ from the first three rules depending on whether we choose b to be true or not, that is, whether we add the fact $b \leftarrow$ or not. Then, on testing them against the integrity constraint expressed by the fourth rule, we see that $\{a, b, c\}$ is indeed a stable model, since it satisfies the integrity constraint, while set $\{a\}$ is not a stable model since checking the integrity constraint fails.

A major intention of austere logic programs is to confine the actual guess and check of an encoding to dedicated components, namely, the choices in C and constraints in I . The definite rules in D help us to analyze and/or extend the solution candidate induced by the facts F and the actual choices in C' . The emerging candidate is then evaluated by the integrity constraints in I . This stresses once more the idea that the extension of a guessed solution candidate should be deterministic; it elaborates the guess but refrains from introducing any ambiguities. This is guaranteed by the definite rules used in austere programs.

Observation 1 *For any austere logic program (F, C, D, I) and $C' \subseteq F_C$, the logic program $F \cup C' \cup D$ has a unique stable model.*

This principle is also in accord with [25], where stratified logic programs are used instead of definite ones (see below).

5 Easy Answer Set Programming

Austere logic programs provide a greatly simplified format that reflects ASP's *guess-define-check* methodology [20] for writing encodings. Their simple structure allows for translating the methodology into an intuitive process that consists of making non-deterministic choices, followed by a deterministic bottom-up computation, and a final consistency check.

In what follows, we want to turn the underlying principles into a modeling methodology for ASP that aims at laypersons. To this end, we leave the propositional setting and aim at full-featured input languages of ASP systems like *clingo* [14] and *dlv* [19]. Accordingly, we shift our attention to predicate symbols rather than propositions and let the terms 'logic program', 'rule', etc. refer to these languages without providing a technical account (cf. [5, 12]). Moreover, we allow for normal rules instead of definite

ones as well as aggregate literals in bodies in order to accommodate the richness of existing ASP modeling languages.

The admission of normal rules comes at the expense of losing control over the origin of multiple or non-existing stable models as well as over a deterministic development of guessed solutions. In fact, the idea of *Easy Answer Set Programming* (ezASP) is to pursue the principles underlying austere logic programs without enforcing them through a severely restricted syntax. However, rather than having the user fully absorb the loss in control, we shift our focus to a well-founded development of ASP encodings, according to which predicates are defined on top of previously defined predicates (or facts). This parallels the structure and the resulting operational semantics of austere logic programs.

To this end, we start by capturing dependencies among predicates [3].

Definition 6. *Let P be a logic program.*

- A predicate symbol p depends upon a predicate symbol q , if there is a rule in P with p on its left-hand side and q on its right-hand side.
If p depends on q and q depends on r , then p depends on r , too.
- The definition of a predicate symbol p is the subset of P consisting of all rules with p on their left-hand side.

We denote the definition of a predicate symbol p in P by $def(p)$ and view integrity constraints as rules defining \perp .

Our next definition makes precise what we mean by a well-founded development of a logic program.⁷

Definition 7. *Let P be a logic program.*

We define a partition (P_1, \dots, P_n) of P as a stratification of P , if

1. $def(p) \subseteq P_i$ for all predicate symbols p and some $i \in \{1, \dots, n\}$ and
2. if p depends on q , $def(p) \subseteq P_i$, and $def(q) \subseteq P_j$ for some $i, j \in \{1, \dots, n\}$, then
 - (a) $i > j$ unless q depends on p , and
 - (b) $i = j$ otherwise

Any normal logic program has such a stratification. One way to see this is that mutually recursive programs can be trivially stratified via a single partition. For instance, this applies to both programs $\{a \leftarrow b, b \leftarrow a\}$ and $\{a \leftarrow \neg b, b \leftarrow \neg a\}$ in which a and b mutually depend upon each other. Accordingly, similar recursive structures in larger programs are confined to single partitions, as required by (2b) above.

With it, we are ready to give shape to the concept of an *easy logic program*.

Definition 8 (Easy logic program). *An easy logic program is a logic program having stratification $(F, C, D_1, \dots, D_n, I)$ such that F is a set of facts, C is a set of choice rules, D_i is a set of normal rules for $i = 1, \dots, n$, and I is a set of integrity constraints.*

As in traditional ASP, we often divide a logic program into facts representing a problem instance and the actual encoding of the problem class. For easy programs, this amounts to separating F from (C, D_1, \dots, D_n, I) .

⁷ The term *stratification* differs from the one used in the literature [3].

Clearly, an austere logic program is also an easy one.

Thus, the program in Listing 1.1 is also an easy logic program having the stratification

$$(\{1, 2, 3\}, \{5\}, \{6, 7\}, \{8, 9, 10\})$$

where each number stands for the rules in the respective line.

Predicates `node/1`, `edge/2`, and `start/1` are only used to form facts or occur in rule bodies. Hence, they do not depend on any other predicates and can be put together in a single component, F . This makes sense since they usually constitute the problem instance. Putting them first reflects that the predicates in the actual encoding usually refer to them. The choices in C provide a solution candidate that is checked by means of the rules in the following components. In our case, the guessed extension of predicate `hc/2` in Line 5 is simply a subset of all edges provided by predicate `edge/2`. Tests for being a path or even a cycle are postponed to the *define-check* part: The rules in $\{6, 7\}$, that is, D_1 , define the auxiliary predicate `reached/1`, and aim at analyzing and/or extending our guessed solution candidate by telling us which nodes are reachable via the instances of `hc/2` from the start node. The actual checks are then conducted by the integrity constraints, I , in the final partition $\{8, 9, 10\}$. At this point, the solution candidate along with all auxiliary atoms are derived and ready to be checked. Line 8 tests whether each node is reached in the solution at hand, while lines 9 and 10 make sure that a valid cycle never enters or leaves any node twice.

Finally, it is instructive to verify that strata $\{5\}$ and $\{6, 7\}$ cannot be reversed or merged. We observe that

- `hc/2` depends on `edge/2` only,

while

- `reached/1` depends on `hc/2`, `edge/2`, `start/1`, and itself,

and no other dependencies. The rules defining `hc/2` and `reached/1` must belong to the same partition, respectively, as required by (2a) above. Thus, $\{5\} \subseteq P_i$ and $\{6, 7\} \subseteq P_j$ for some i, j . Because `reached/1` depends on `hc/2` and not vice versa, we get $i < j$. This dependency rules out an inverse arrangement, and the fact that it is not symmetric excludes a joint membership of both definitions in the same partition, as stipulated by (2b) above.

5.1 Modeling Methodology

The backbone of easy ASP's modeling methodology is the structure imposed on its programs in Definition 8. This allows us to encode problems by building concepts on top of each other. Also, its structure allows for staying in full tune with ASP's *guess-define-check* methodology [20] by offering well-defined spots for all three parts.

Easy logic programs tolerate normal rules in order to encompass full-featured ASP modeling languages. Consequently, the interplay of the guess, define, and check parts of an easy logic program defies any control. To tame this opening, we propose to carry over Observation 1 to easy logic programs: For any easy logic program

$(F, C, D_1, \dots, D_n, I)$ and $C' \subseteq F_C$, the logic program $F \cup C' \cup D_1 \cup \dots \cup D_n$ should have a unique stable model. Even better if this can be obtained in a deterministic way.

This leads us to the following advice on easy ASP modeling:

1. Compartmentalize a logic program into facts, F , choice rules, C , normal rules, $D_1 \cup \dots \cup D_n$, and integrity constraints I , such that the overall logic program has stratification $(F, C, D_1, \dots, D_n, I)$.
2. Aim at defining one predicate per stratum D_i and avoid cycles within each D_i for $i = 1, \dots, n$.
3. Ensure that $F \cup C' \cup D_1 \cup \dots \cup D_n$ has a unique stable model for any $C' \subseteq F_C$.

While the first two conditions have a syntactic nature and can thus be checked automatically, the last one refers to semantics and, to the best of our knowledge, has only sufficient but no known necessary syntactic counterparts. One is to restrict $D_1 \cup \dots \cup D_n$ to definite rules as in austere programs, the other is to use stratified negation, as proposed in [25] and detailed in the next section.

Our favorite is to stipulate that $F \cup C' \cup D_1 \cup \dots \cup D_n$ has a total well-founded model [28] for any $C' \subseteq F_C$ but unfortunately, we are unaware of any syntactic class of logic programs warranting this condition beyond the ones mentioned above.

5.2 Stratified Negation

The purpose of stratified negation is to eliminate the (problematic) case of recursion through negation. What makes this type of recursion problematic is that it may eliminate stable models and that the actual source may be spread over several rules. To give some concise examples, consider the programs $\{a \leftarrow \neg a\}$ and $\{a \leftarrow \neg b, b \leftarrow \neg c, c \leftarrow \neg a\}$ admitting no stable models. Following the dependencies in both examples, we count one and three dependencies, respectively, all of which pass through negated body literals. More generally, cyclic dependencies traversing an odd number of negated body literals (not necessarily consecutively) are known sources of incoherence. Conversely, an even number of such occurrences on a cycle is not harmful but spawns alternatives, usually manifested in multiple stable models. To see this, consider the program $\{a \leftarrow \neg b, b \leftarrow \neg a\}$ producing two stable models. Neither type of rule interaction is admitted in austere logic programs. Rather the idea is to confine the sources of multiple and eliminated stable models to dedicated components, namely, choices and integrity constraints. The same idea was put forward by Niemelä in [25] yet by admitting a more general setting than definite rules by advocating the concept of stratified negation.

To eliminate the discussed cyclic constellations, stratified negation imposes an additional constraint on the stratification of a logic program: Given the prerequisites of Definition 7, we define:

3. If a predicate symbol q occurs in a negative body literal of a rule in P_i , then $def(q) \subseteq P_j$ for some $j < i$.

In other words, while the definitions of predicates appearing positively in rule bodies may appear in a lower or equal partition, the ones of negatively occurring predicates are restricted to lower components. Although this condition tolerates positive recursion as

in $\{a \leftarrow b, b \leftarrow a\}$, it rules out negative recursion as in the above programs. Since using programs with stratified negation rather than definite programs generalizes austere logic programs, their combination with choices and integrity constraints is also as expressive as full ASP [25].

An example of stratified negation can be found in Listing 1.3. The negative literal in Line 5 refers to a predicate defined—beforehand—in Line 8.

An attractive feature of normal logic programs with stratified negation is that they yield a unique stable model, just as with austere programs (cf. Observation 1). Hence, they provide an interesting generalization of definite rules maintaining the property of deterministically extending guessed solution candidates.

5.3 Complex Constraints

As mentioned, we aim at accommodating complex language constructs as aggregates in order to leverage the full expressiveness of ASP’s modeling languages. For instance, we may replace lines 9 and 10 in Listing 1.1 by

```

9  :- { hc(U,V) } >= 2, node(U).
10 :- { hc(U,V) } >= 2, node(V).
    
```

without violating its stratification.

More generally, a rule with an aggregate ‘ $\#op\{l_1, \dots, l_m\} \prec k$ ’ in the consequent can be represented with choice rules along with an integrity constraint, as shown in [27]. That is, we can replace any rule of form

$$\#op\{l_1, \dots, l_m\} \prec k \leftarrow l_{m+1} \wedge \dots \wedge l_n$$

by⁸

$$\begin{aligned} \{l_i\} \leftarrow l_{m+1} \wedge \dots \wedge l_n & \quad \text{for } i = 1, \dots, m \text{ and} \\ \perp \leftarrow \neg(\#op\{l_1, \dots, l_m\} \prec k) \wedge l_{m+1} \wedge \dots \wedge l_n. & \end{aligned}$$

This allows us to integrate aggregate literals into easy logic programs without sacrificing expressiveness.

In fact, many encodings build upon restricted choices that are easily eliminated by such a transformation. A very simple example is graph coloring. Assume a problem instance is given in terms of facts `node/1`, `edge/2`, and `color/1`. A corresponding encoding is given by the following two rules:

```

1  { assign(X,C) : color(C) } = 1 :- node(X).
2  :- edge(X,Y), assign(X,C), assign(Y,C).
    
```

Note that the aggregate in the consequent of Line 1 is a shortcut for a `#count` aggregate.

To eliminate the restricted choice from the consequent in Line 1, we may apply the above transformation to obtain the following easy encoding:

⁸ In practice, a set of such choice rules can be represented by a single one of form $\{l_1, \dots, l_m\} \leftarrow l_{m+1} \wedge \dots \wedge l_n$.

```

1 { assign(X,C) } :- node(X), color(C).
2 :- not { assign(X,C) : color(C) } = 1, node(X).
3 :- edge(X,Y), assign(X,C), assign(Y,C).

```

Given some set of facts, F , this encoding amounts to the easy logic programs $(F, \{1\}, \{2\}, \{3\})$.

The decomposition into a choice and its restriction may appear unnecessary to the experienced ASP modeler. However, we feel that such a separation adds clarity and is preferable to language constructs combining several aspects, at least for ASP beginners. Also, it may be worth noting that this decomposition is done anyway by an ASP system and hence brings about no performance loss.

Two further examples of easy logic programs are given in Listing 1.2 and 1.3, solving the Queens and the Tower-of-Hanoi puzzles both with parameter n .⁹ While the

```

1 { queen(1..n,1..n) }.
2
3 d1(I,J,I-J+n) :- I = 1..n, J = 1..n.
4 d2(I,J,I+J-1) :- I = 1..n, J = 1..n.
5
6 :- { queen(I,1..n) } != 1, I = 1..n.
7 :- { queen(1..n,J) } != 1, J = 1..n.
8
9 :- { queen(I,J) : d1(I,J,D) } > 1, D=1..n*2-1.
10 :- { queen(I,J) : d2(I,J,D) } > 1, D=1..n*2-1.

```

Listing 1.2. An easy logic program for the n -Queens puzzle

easy logic program for the n -Queens puzzle has the format

$$(\emptyset, \{1\}, \{3, 4\}, \{6, 7\}, \{9, 10\}),$$

the one for the Tower-of-Hanoi puzzle can be partitioned into

$$(\{1, 2, 3, 4\}, \{6\}, \{8\}, \{10, 11, 12\}, \{14, 15\}, \{17, 19, 20, 21, 23\}) .$$

5.4 Limitations

The methodology of ezASP has its limits. For instance, sometimes it is convenient to make choices depending on previous choices. Examples of this are the combination of routing and scheduling, as in train scheduling [1], or the formalization of frame axioms in (multi-valued) planning advocated in [18]. Another type of encodings escaping our methodology occurs in meta programming, in which usually a single predicate, like `holds`, is used and atoms are represented as its arguments. Thus, for applying the

⁹ This parameter is either added from the command line via option `--const` or a default added via directive `#const` (see [13] for details).

```

1  peg(a;b;c) .
2  disk(1..4) .
3  init_on(1..4,a) .
4  goal_on(1..4,c) .
5
6  { move(D,P,T) : disk(D), peg(P) } :- T = 1..n.
7
8  move(D,T)          :- move(D,_,T) .
9
10 on(D,P,0)          :- init_on(D,P) .
11 on(D,P,T)          :- move(D,P,T) .
12 on(D,P,T+1)        :- on(D,P,T), not move(D,T+1), T < n.
13
14 blocked(D-1,P,T+1) :- on(D,P,T), T < n.
15 blocked(D-1,P,T)   :- blocked(D,P,T), disk(D) .
16
17 :- { move(D,P,T) : disk(D), peg(P) } != 1, T = 1..n.
18
19 :- move(D,P,T), blocked(D-1,P,T) .
20 :- move(D,T), on(D,P,T-1), blocked(D,P,T) .
21 :- not 1 { on(D,P,T) } 1, disk(D), T = 1..n.
22
23 :- goal_on(D,P), not on(D,P,n) .

```

Listing 1.3. An easy logic program for a Towers-of-Hanoi puzzle (for plans of length n)

ezASP methodology, one had to refine the concept of stratification to access the term level in order to capture the underlying structure of the program. And finally, formalizations of planning and reasoning about actions involve the formalization of effect and inertia laws that are usually self-referential on the predicate level (sometimes resolved on the term level, through situation terms or time stamps). A typical example of circular inertia laws is the following:

```

holds(F,T) :- holds(F,T-1), not -holds(F,T) .
-holds(F,T) :- -holds(F,T-1), not holds(F,T) .

```

Here, ‘-’ denotes classical negation, and F and T stand for (reified) atoms and time points. On the other hand, the sophistication of the given examples illustrates that they are usually not addressed by beginners but rather experts in ASP for which the strict adherence to ezASP is less necessary.

6 Related Work

Apart from advocating the idea illustrated in the introduction, Ilkka Niemelä also showed in [25] that negative body literals can be replaced by a new atom for which a choice needs to be made whether to include it in the model or not; and such that a model cannot contain both the new atom and the atom of the replaced literal but one

of them needs to be included. This technique amounts exactly to the transformation in Definition 2 and traces back to Abductive logic programming [7,9]. Indeed, it was already shown in [16] that for DATALOG queries the expressive power of stable model semantics can be achieved via stratified negation and choices.

We elaborated upon this idea in several ways. First, we have shown that the full expressiveness of normal logic programs can even be achieved with definite rules rather than normal ones with stratified negation. Second, we have provided a strong equivalence result that allows for applying the transformation in Definition 2 to selected rules only. Third, we have generalized the idea by means of the logic of Here-and-There, which made it applicable to other fragments of logic programs. And finally, this investigation has revealed that the roots of the idea lie in the logical principle of extension by definition.

Over the last decades many more related ideas were presented in the literature. For instance, in [10], normal logic programs are translated into positive disjunctive programs by introducing new atoms for negative literals. Also, strong negation is usually compiled away via the introduction of new atoms along with integrity constraints excluding that both the original atom and the atom representing its strong negation hold [15]. The principle of extension by definition was also used in [11] to prove properties about programs with nested expressions. EzASP is closely related to the paradigm of IDP [6], where the program parts F , C and I are expressed in first-order logic, while the D_i 's form inductive definitions. Finally, in [8], the authors propose an informal semantics for logic programs based on the guess-define-check methodology, that are similar to the easy logic programs that we introduce in this paper.

7 Conclusion

We have revisited an old idea from the literature on logic programming under stable model semantics and elaborated upon it in several ways. We started by tracing it back to the principle of extension by definition. The resulting formalization in the setting of the logic of Here-and-there provides us with a logical framework that can be instantiated in various ways. Along these lines, we have shown that normal logic programs can be reduced to choices, definite rules, and integrity constraints, while keeping the same expressiveness as the original program. A major advantage of this austere format is that it confines non-determinism and incoherence to well-defined spots in the program. The resulting class of austere logic programs could play a similar role in ASP as formulas in conjunctive normal form in classical logic.

Drawing on the properties observed on austere logic program, we put forward the modeling methodology of ezASP. The idea is to compensate for the lacking guarantees provided by the restricted format of austere programs by following a sequential structure when expressing a problem in terms of a logic program. This makes use of the well-known concept of stratification to refine ASP's traditional guess-define-check methodology. Although the ordering of rules may seem to disagree with the holy grail of full declarativeness, we evidence its great value in introducing beginners to ASP. Also, many encodings by experienced ASP users follow the very same pattern.

Moreover, the ezASP paradigm aligns very well with that of *achievements* [21] that aims not only at easily understandable but moreover provably correct programs. To

this end, formal properties are asserted in between a listing of rules to express what has been achieved up to that point. Extending ezASP with achievements and automatically guiding the program development with ASP verifiers, like *anthem* [22], appears to us as a highly interesting avenue of future research. In this context, it will also be interesting to consider the components of an easy logic program as modules with an attached input-output specification, so that the meaning of the overall program emerges from the composition of all components. This would allow for successive refinements of programs' components, while maintaining their specification.

References

1. Abels, D., Jordi, J., Ostrowski, M., Schaub, T., Toletti, A., Wanko, P.: Train scheduling with hybrid ASP. In: Balduccini et al. [4], pp. 3–17
2. Aguado, F., Cabalar, P., Fandinno, J., Pearce, D., Pérez, G., Vidal, C.: Forgetting auxiliary atoms in forks. *Artif. Intell.* **275**, 575–601 (2019)
3. Apt, K., Blair, H., Walker, A.: Towards a theory of declarative knowledge. In: Minker, J. (ed.) *Foundations of Deductive Databases and Logic Programming*, chap. 2, pp. 89–148. Morgan Kaufmann Publishers (1987)
4. Balduccini, M., Lierler, Y., Woltran, S. (eds.): Logic programming and nonmonotonic reasoning. In: *Proceedings of the Fifteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019)*, LNAI, vol. 11481. Springer, Cham (2019). <https://doi.org/10.1007/978-3-030-20528-7>
5. Calimeri, F., et al.: ASP-Core-2 input language format. *Theory Pract. Log. Program.* **20**(2), 294–309 (2019)
6. De Cat, B., Bogaerts, B., Bruynooghe, M., Janssens, G., Denecker, M.: Predicate logic as a modeling language: the IDP system, pp. 121–177. ACM/Morgan, Claypool (2018)
7. Denecker, Marc, Kakas, Antonis: Abduction in logic programming. In: Kakas, Antonis C., Sadri, Fariba (eds.) *Computational Logic: Logic Programming and Beyond*. LNCS (LNAI), vol. 2407, pp. 402–436. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45628-7_16
8. Denecker, M., Lierler, Y., Truszczyński, M., Vennekens, J.: The informal semantics of answer set programming: A Tarskian perspective. *CoRR* abs/1901.09125 (2019). <http://arxiv.org/abs/1901.09125>
9. Eshghi, K., Kowalski, R.: Abduction compared with negation by failure. In: Levi, G., Martelli, M. (eds.) *Proceedings of the Sixth International Conference on Logic Programming (ICLP 1989)*, pp. 234–255. MIT Press (1989)
10. Fernández, J., Lobo, J., Minker, J., Subrahmanian, V.: Disjunctive LP + integrity constraints = stable model semantics. *Ann. Math. Artif. Intell.* **8**(3–4), 449–474 (1993)
11. Ferraris, P., Lifschitz, V.: Weight constraints as nested expressions. *Theory Pract. Log. Program.* **5**(1–2), 45–74 (2005)
12. Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., Schaub, T.: Abstract Gringo. *Theory Pract. Log. Program.* **15**(4–5), 449–463 (2015). <https://doi.org/10.1017/S1471068415000150>
13. Gebser, M., et al.: *Potassco user guide*. University of Potsdam, 2 edn. (2015). <http://potassco.org>
14. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.* **19**(1), 27–82 (2019). <https://doi.org/10.1017/S1471068418000054>

15. Gelfond, M., Lifschitz, V.: Logic programs with classical negation. In: Warren, D., Szeredi, P. (eds.) *Proceedings of the Seventh International Conference on Logic Programming (ICLP 1990)*, pp. 579–597. MIT Press (1990)
16. Greco, S., Saccà, D., Zaniolo, C.: Extending stratified datalog to capture complexity classes ranging from P to QH. *Acta Informatica* **37**(10), 699–725 (2001)
17. Heyting, A.: Die formalen Regeln der intuitionistischen Logik. In: *Sitzungsberichte der Preussischen Akademie der Wissenschaften*, pp. 42–56. Deutsche Akademie der Wissenschaften zu Berlin (1930)
18. Lee, J., Lifschitz, V., Yang, F.: Action language BC: preliminary report. In: Rossi, F. (ed.) *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence (IJCAI 2013)*, pp. 983–989. IJCAI/AAAI Press (2013)
19. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.* **7**(3), 499–562 (2006)
20. Lifschitz, V.: Answer set programming and plan generation. *Artif. Intell.* **138**(1–2), 39–54 (2002)
21. Lifschitz, V.: Achievements in answer set programming. *Theory Pract. Log. Program.* **17**(5–6), 961–973 (2017)
22. Lifschitz, V., Lühne, P., Schaub, T.: Verifying strong equivalence of programs in the input language of GRINGO. In: Balduccini et al. [4], pp. 270–283
23. Lloyd, J.W.: *Foundations of Logic Programming*. Springer, Heidelberg (1987). <https://doi.org/10.1007/978-3-642-83189-8>
24. McCarthy, J.: Elaboration tolerance (1998). <http://jmc.stanford.edu/articles/elaboration/elaboration.pdf>
25. Niemelä, I.: Answer set programming without unstratified negation. In: Garcia de la Banda, M., Pontelli, E. (eds.) *ICLP 2008*. LNCS, vol. 5366, pp. 88–92. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89982-2_15
26. Pearce, D.: Equilibrium logic. *Ann. Math. Artif. Intell.* **47**(1–2), 3–41 (2006). <https://doi.org/10.1007/s10472-006-9028-z>
27. Simons, P., Niemelä, I., Sooinen, T.: Extending and implementing the stable model semantics. *Artif. Intell.* **138**(1–2), 181–234 (2002)
28. Van Gelder, A., Ross, K., Schlipf, J.: The well-founded semantics for general logic programs. *J. ACM* **38**(3), 620–650 (1991)